

Python and the XML Document Object Model (DOM) with 4Suite



John W. Shipman

2007-09-17 20:10



Table of Contents

1. Introduction	3
1.1. How to get this publication	3
1.2. When not to use the DOM	3
1.3. Unicode good, strings risky	4
2. Terms you should know	4
2.1. URI	4
2.2. URL and URN	4
2.3. Absolute URI	4
2.4. Relative URI	5
2.5. Base URI	5
2.6. Namespace URI	5
2.7. QName: Qualified name	6
2.8. Namespace prefix	6
2.9. Local name	7
3. Reading an XML document in Python	7
3.1. A quick and dirty document reader	7
3.2. A full-featured reader	7
4. The structure of a DOM tree	8
5. The Node class	8
5.1. Node.appendChild()	10
5.2. Node.cloneNode()	10
5.3. Node.hasChildNodes()	10
5.4. Node.insertBefore()	11
5.5. Node.isSameNode()	11
5.6. Node.removeChild()	11
5.7. Node.replaceChild()	11
5.8. Node.xpath()	11
6. The Document class	12
7. The Element class	12
7.1. Element.getAttributeNS()	12
7.2. Element.getAttributeNodeNS()	13
7.3. Element.hasAttributeNS()	13
7.4. Element.removeAttributeNS()	13
7.5. Element.setAttributeNS()	13
7.6. Element.setAttributeNodeNS()	14
8. The Attr class	14

9. The <code>CharacterData</code> classes: <code>Text</code> , <code>CDATA</code> , and <code>Comment</code>	14
9.1. <code>CharacterData.appendData()</code>	14
9.2. <code>CharacterData.deleteData()</code>	14
9.3. <code>CharacterData.insertData()</code>	15
9.4. <code>CharacterData.replaceData()</code>	15
9.5. <code>CharacterData.substringData()</code>	15
10. The <code>DOMImplementation</code> object	15
10.1. <code>DOMImplementation.createDocument()</code>	16
10.2. <code>DOMImplementation.createRootNode()</code>	16
11. Printing a document	17
12. Creating a document from scratch: factory methods	17
12.1. <code>Document.createElementNS()</code>	18
12.2. <code>Document.createTextNode()</code>	18
12.3. <code>Document.createProcessingInstruction()</code>	19
12.4. <code>Document.createComment()</code>	19
12.5. <code>Document.createDocumentFragment()</code>	19
12.6. An example of document creation	19
13. <code>xml4create.py</code> : A more Pythonic module for XML file creation	20
13.1. The <code>DocumentType</code> class	21
13.2. The <code>Document</code> class	21
13.3. The <code>Element</code> class	22
13.4. The <code>Text</code> class	23
13.5. The <code>Comment</code> class	23
13.6. The <code>DocumentFragment</code> class	24
13.7. A small example: XHTML page generation	24
14. <code>xml4create.py</code> : The source code	25
14.1. Prologue to <code>xml4create.py</code>	26
14.2. The <code>DocumentType</code> class	26
14.3. The <code>Document</code> class	27
14.4. <code>Document.splitQName():</code> Process a qualified name	28
14.5. <code>Document.serialize()</code>	30
14.6. <code>Document.write()</code>	30
14.7. The <code>Element</code> class	30
14.8. <code>Element.__init__()</code>	31
14.9. <code>Element.__wrapRoot():</code> Wrap the root element	32
14.10. <code>Element.__setAttr():</code> Set up one attribute	33
14.11. <code>Element.__newElement():</code> Element child of an element	34
14.12. <code>Element.__setitem__()</code>	35
14.13. <code>Element.update():</code> Copy XML attributes to the element	35
14.14. The <code>Text</code> class	36
14.15. <code>Text.__init__()</code>	37
14.16. The <code>Comment</code> class	37
14.17. The <code>DocumentFragment</code> class	38
14.18. <code>DocumentFragment.serialize()</code>	39
14.19. <code>DocumentFragment.write()</code>	39
15. Test driver for multiple namespaces: <code>crens</code>	39
16. Test driver for a document fragment: <code>crefrag</code>	40

1. Introduction

The Python programming language is a fine vehicle for writing programs that process or generate XML files. This document is a quick reference to the major features of Python's interface to the XML Document Object Model (DOM), which provides for a tree-like data structure that represents an XML document. We won't cover every feature; this is just a selection of commonly used techniques.

The reader is assumed to be generally familiar with the Python language and with the structure of XML documents. For more information, see these references:

- The `python.org` page¹ is the central site for the Python language. See also TCC's Python page².
- For an introduction to Python's approach to the DOM, see the *Python Library Reference* page, "`xml.dom`—The Document Object Model API"³.
- For help on XML, see TCC's XML page.⁴
- This document assumes you have the 4Suite package installed. For information and downloads, see the Fourthought page⁵.

1.1. How to get this publication

This publication is available in Web form⁶ and also as a PDF document⁷. Please forward any comments to `tcc-doc@nmt.edu`.

This document also presents a number of Python source files in literate programming style: a useful module and a few short test drivers demonstrating its use. These files are available online:

- `xml4create.py`⁸: See Section 13, "`xml4create.py`: A more Pythonic module for XML file creation" (p. 20).
- `xmlcreatetest`⁹: Simple test driver for `xml4create.py`.
- `crens`¹⁰: Test driver demonstrating output in multiple namespaces.
- `crefrag`¹¹: Test driver demonstrating creation of a document fragment.

1.2. When not to use the DOM

For most XML processing, the Document Object Model is a convenient way to read or write XML files. However, there are some situations where other techniques may be necessary.

- A DOM tree is resident entirely in memory. Obviously, if you are reading a five-gigabyte XML file, it probably won't fit.
- Building a DOM tree from a document can be a slow process for larger files.

¹ <http://www.python.org/>

² <http://www.nmt.edu/tcc/help/lang/python/>

³ <http://docs.python.org/lib/module-xml.dom.html>

⁴ <http://www.nmt.edu/tcc/help/xml/>

⁵ <http://4suite.org/>

⁶ <http://www.nmt.edu/tcc/help/pubs/pyxml4/>

⁷ <http://www.nmt.edu/tcc/help/pubs/pyxml4/pyxml4.pdf>

⁸ <http://www.nmt.edu/tcc/help/pubs/pyxml4/xml4create.py>

⁹ <http://www.nmt.edu/tcc/help/pubs/pyxml4/xmlcreatetest>

¹⁰ <http://www.nmt.edu/tcc/help/pubs/pyxml4/crens>

¹¹ <http://www.nmt.edu/tcc/help/pubs/pyxml4/crefrag>

If memory size or performance are issues, refer to the SAX processing model, which is outside the scope of this document. Here is a good resource:

Jones, Christopher A., and Fred L. Drake, Jr. Python & XML. O'Reilly, 2002, ISBN 0-596-00128-2.

1.3. Unicode good, strings risky

Throughout the descriptions of these interfaces, we describe a number of arguments and results as “strings”. It is safest to use Python unicode strings throughout these interfaces. Regular Python strings of class `str` will be converted to Unicode automatically, but the automatic conversion may not do the right thing.

2. Terms you should know

Let's review some terms used throughout this document. Some of these have been around a while, but their meanings have mutated or been clarified recently.

2.1. URI

URI, for Uniform Resource Identifier, is a string of characters that can be used to identify a resource. It can be either or both of:

- a *locator* that specifies how to find something on the Internet, and/or:
- A name that uniquely identifies a resource. That resource may or may not be located on the Internet.

For example, the URI “`http://www.nmt.edu/`” is New Mexico Tech's World Wide Web homepage. That URI is both a locator (it tells your browser where to find the page) and the name of the resource.

However, the URI “`urn:ISBN 0-201-38595-3`” identifies a physical book. This is a resource name, but not a locator: it does not tell you how to get the book.

For a good discussion of URIs, see the *Wikipedia* article¹².

2.2. URL and URN

Two older terms are becoming deprecated:

URL

Uniform Resource Locator. This is a URI that implies that the resource is online, and there is a standard protocol for translating such URIs to Internet addresses.

URN

Uniform Resource Name. A URN identifies a resource, but that resource may not necessarily be online.

2.3. Absolute URI

A valid absolute URI must have three parts:

¹² http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

1. A *scheme name* that defines what kind of URI it is. The most common example is `http`, which specifies the Hypertext Transfer Protocol used by the World Wide Web.
2. A colon character, “:”
3. A *scheme-specific part* follows the colon. The syntax of this part depends on the scheme name.

For example, in the URI “`http://www.nmt.edu/tcc/`”, the scheme name is “`http`” and the scheme-specific part is “`//www.nmt.edu/tcc/`”.

2.4. Relative URI

It is cumbersome to use a full, absolute URI for every reference to a resource. Consequently, you can use a *relative URI* in many locations.

For example, a document located at “`http://www.nmt.edu/`” may refer to another document located at “`http://www.nmt.edu/tcc/homepage.html`” using that absolute URL. Alternately, it may use the relative URL “`tcc/homepage.html`”.

A relative URI must always be evaluated in the context of an absolute *base URI*. In the example above, the relative URI is evaluated in the context of the base URI “`http://www.nmt.edu/`”.

For complete details of URIs and the process of converting a relative URI to absolute form, see *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*¹³.

2.5. Base URI

The *base URI* of a document is usually the URI where that document is located, but a document can use the special `xml:base` attribute to specify some other value.

For example, the `XInclude` standard allows you to refer to other files, but relative URI references in an `XInclude` element are evaluated relative to the base URI of the reference.

2.6. Namespace URI

Any XML element or attribute may belong to a *namespace*, that is, a set of elements and attributes that belong to a single document type.

Most XML document types are defined by a *schema* that not only enumerates all the element and attribute names in that document type, but also defines the rules for structuring the document: which elements may be inside which other elements, and which attributes may occur in which elements.

Throughout the present document, we use the term *namespace URI* to mean a URI that uniquely identifies a namespace.

For example, the XHTML 1.0 document type is associated with this namespace URI:

```
http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd
```

In this case, there actually is a DTD (Document Type Definition) schema located on the Web at that URI. However, a namespace URI may be only an identifier, and it may or may not be function as a location.

¹³ <http://www.ietf.org/rfc/rfc3986.txt?number=3986>

2.7. QName: Qualified name

A document may contain elements and attributes from more than one namespace URI. Because namespace URIs can be quite lengthy, it is cumbersome to include the complete namespace URI in each element or attribute name.

XML allows a shorthand notation to simplify assigning names to namespaces. You may invent a short *namespace prefix* and associate it with a namespace URI.

For example, suppose your document contains names from both the XSLT and XHTML namespaces. In this situation, it is customary to associate the namespace prefix “xsl” with XSLT. You might use prefix “html” to refer to the XHTML namespace.

A *qualified name* has three parts:

1. A namespace prefix.
2. A colon character, “:”.
3. A *local name* that gives the name of the element or attribute within that namespace.

For example, here is a fragment illustrating the use of qualified names:

```
<xsl:template match="separator">
  <html:hr/>
</xsl:template>
```

Element `template` is in the XSLT namespace, and element `hr` is in the HTML namespace.

You can also use an *unqualified name*, which is just a local name without the namespace prefix. The namespace of such a name is called the *default namespace*.

The association between a namespace prefix and a namespace URI is made with an `xmlns` attribute located in some element. This attribute can have two forms:

- An attribute named `xmlns='nsURI'` defines the namespace URI of the default namespace. This namespace is associated with all elements and attributes that have unqualified names.
- An attribute named `xmlns:prefix='nsURI'` associates the given *prefix* with the namespace URI *nsURI*.

Here is an example of a complete document using elements from two namespaces:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
  version="1.0">
  <xsl:template match="foo">
    <hr/>
  </xsl:template>
</xsl:stylesheet>
```

In this example, the `template` element is in the XSLT namespace. The `hr` element is in the default namespace, which is associated with the XHTML namespace URI.

2.8. Namespace prefix

See Section 2.7, “QName: Qualified name” (p. 6).

2.9. Local name

See Section 2.7, “qName: Qualified name” (p. 6).

3. Reading an XML document in Python

To extract information from an XML document, you'll need to read it and convert into the DOM tree form. There is an easy way to do this, and a full-featured way to do it. In order to select between these methods, it is necessary to think about whether the document needs a correct *base URI* (see Section 2.5, “Base URI” (p. 5)).

Consequently:

- If your document has no relative URI references, see Section 3.1, “A quick and dirty document reader” (p. 7).
- If your document has relative URI references, see Section 3.2, “A full-featured reader” (p. 7).

3.1. A quick and dirty document reader

If you don't need to supply a correct base URI, this technique gives you a DOM document object that represents an XML source document in any of four forms:

- A string that contains the entire document, e.g., “<dog-list><dog breed='basset' sex='m'>Rover</dog ></dog-list >”
- A file containing the document, as a readable file object.
- A string that names the file containing the document.
- The URI of the document, if it is available at that location.

To use this technique, first import the relevant modules:

```
from Ft.Xml import Parse
```

Then, to transform an XML document into a DOM tree:

```
doc = Parse ( source )
```

where *source* is any of: a string containing the document, a stream from which to read the document, the name of the document file, or the URI of the document.

The `Parse()` function returns a `DOM Document` node, that is, the root of the document tree. For further information on the structure of this tree, see Section 4, “The structure of a DOM tree” (p. 8).

If the *source* does not exist, this function will raise `Ft.Lib.UriException`. If it exists but is not well-formed, `Parse()` will raise `Ft.Xml.ReaderException`.

3.2. A full-featured reader

In the cases where your document needs a correct base URI, use this form of `import`:

```
from Ft.Xml.Domlette import NonvalidatingReader
```

In case you're wondering, there is also a `ValidatingReader` that uses the document's `<!DOCTYPE>` declaration to find its schema, and validates the document against the schema. We'll assume that validation is taken care of elsewhere.

Then, to turn that document into a DOM tree, use one of these methods:

`NonvalidatingReader.parseURI (uri)`

Read the document from a given URI. The `uri` argument specifies the document's location as well as its base URI.

`NonvalidatingReader.parseStream (f, uri)`

Read the document from a readable file object `f`. The `uri` argument must be the document's base URI.

`NonvalidatingReader.parseString (s, uri)`

The `s` argument is a string containing the entire document. The `uri` argument must be the document's base URI.

Each of these reader functions returns a `Document` instance.

4. The structure of a DOM tree

An XML document, represented as a DOM tree, is a data structure made of *nodes*. Each node is an object that inherits from a fundamental `Node` class. We'll start by discussing the attributes and methods of the `Node` class, then proceed to discuss the different subclasses that represent elements, attributes, and the other objects in an XML document.

A DOM tree doesn't have to be a static, fixed entity. You can read a document as described above, and then modify it and write it back out. Your program can also create an entire new document; see Section 12, "Creating a document from scratch: factory methods" (p. 17).

5. The Node class

All the different kinds of nodes in a DOM tree share these attributes from `class Node`:

`.nodeType`

The type of the node as an integer constant.

`.ELEMENT_NODE`, `.ATTRIBUTE_NODE`, etc.

All nodes have a set of constant attributes that define the `.nodeType` values for the various node types. Here is a table showing all these constants:

<code>.ATTRIBUTE_NODE</code>	Represents an attribute of an XML element.
<code>.CDATA_SECTION_NODE</code>	Represents a CDATA section.
<code>.COMMENT_NODE</code>	Represents a comment (<code><!-- ... --></code>).
<code>.DOCUMENT_FRAGMENT_NODE</code>	Represents a fragment of a document.
<code>.DOCUMENT_NODE</code>	Represents an entire document.
<code>.DOCUMENT_TYPE_NODE</code>	Represents a document type identifier (<code><!DOCTYPE ... ></code>).
<code>.ELEMENT_NODE</code>	Represents an XML element.
<code>.ENTITY_NODE</code>	Represents an entity.

.ENTITY_REFERENCE_NODE	Represents a reference to an entity (& . . . ;).
.PROCESSING_INSTRUCTION_NODE	Represents a processing instruction (<? . . . ?>).
.TEXT_NODE	Represents some text.

.nodeName

The name of the node.

- In an element node, this is the element name (e.g., "chapter" for a <chapter> . . . </chapter> element).
- For an Attr node, it is the attribute name.
- For a document type node, it is the name of the document type.
- For a processing instruction, it is the target name.

.nodeValue

The value of the node.

- For an attribute node, it is the attribute's value as a string.
- For a CDATA section, comment, or text node, it is the text inside the CDATA section, comment, or text section.
- For a processing instruction, it is the content part.

.attributes

Used only for Element nodes, its value is a Python dictionary containing the element's attributes.

In this dictionary, the key of each attribute is a 2-tuple (*nsURI*, *localName*), where:

nsURI

is the attribute's namespace URI, or None for the default namespace.

localName

is the attribute's unqualified name.

The corresponding value for that key is the attribute value as a Unicode string.

For example, for an attribute expressed as `class='alarm'`, the key would be `(None, u'class')` and the value would be `u'alarm'`.

.childNodes

If the element has children, this attribute is a list of Node objects containing its children in document order.

For Element nodes, this is a list of its child elements; its attributes are not considered children in this sense.

For Document or DocumentFragment nodes, the children might include comments, document types, and processing instructions, as well as an element child that is the root of the XML document.

.firstChild

If the element has children, this attribute will be set to the first child.

.lastChild

If the element has children, this attribute will be the last child.

.localName

For Element or Attr nodes, holds the local part of a fully qualified name. For example, if the element name is `"xsl:template"`, the `.localName` attribute is `"template"`.

.namespaceURI

For names with a namespace prefix, holds the URI associated with the namespace; otherwise `None`. Compare the `.prefix` attribute.

.nextSibling

The next child of the same parent, if any, otherwise `None`.

.ownerDocument

For every node in the tree, this attribute points at the `Document` node that roots the tree.

.parentNode

The element's parent node, or `None` if the element is the root of the tree.

.prefix

The namespace prefix of the element, or `None` if it doesn't have one. For example, if in this document namespace prefix `xsl:` is defined by

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

then the `.prefix` attribute of an `xsl:template` element's `.prefix` attribute would be `"xsl"`, and its `.namespaceURI` attribute would be `"http://www.w3.org/1999/XSL/Transform"`.

.previousSibling

The element's parent's previous child if there is one, otherwise `None`.

Methods on `Node` objects include:

5.1. Node.appendChild()

Once you have created a node `newChild` (e.g., with Section 12.1, “`Document.createElementNS()`” (p. 18)), use this method to attach it to a parent node `N`:

```
N.appendChild(newChild)
```

If `N` already had some children, the new node will go after the existing children.

5.2. Node.cloneNode()

Use this method to create a duplicate of a node `N`:

```
N.cloneNode(deep)
```

The returned value is the newly created copy.

If the `deep` argument is false, you get just a copy of `N`. If `deep` is true, you get a copy of `N`, complete with copies of all its descendant nodes attached.

5.3. Node.hasChildNodes()

To test whether a node `N` has any child nodes:

```
N.hasChildNodes()
```

Returns true if there are child nodes, false otherwise.

5.4. Node.insertBefore()

Use this method to insert a child node *new* at a position before an existing given child node *old*, under a parent *N*:

```
N.insertBefore(new, old)
```

5.5. Node.isSameNode()

This predicate tests whether two nodes are the same identical node:

```
N.isSameNode ( otherNode )
```

If *otherNode* is the same node as *N*, this method returns true; otherwise it returns false.

5.6. Node.removeChild()

Use this method to remove a specific child *old* of a parent node *N*:

```
N.removeChild(old)
```

5.7. Node.replaceChild()

Use this method to replace an existing child *old* with a new child *new* under a parent node *N*:

```
N.replaceChild(new, old)
```

5.8. Node.xpath()

It is often convenient to use XPath expressions to quickly locate nodes within a tree. For a description of the XPath language, see *XSLT Reference*¹⁴ or the standard, *XML Path Language (XPath)*¹⁵.

This method allows you to evaluate an XPath expression relative to a context node *N*. The *expr* argument is an XPath expression string.

```
N.xpath(expr, nsMap )
```

The `.xpath` method returns the result of the evaluation of the XPath expression. For example, if the XPath expression produces a node list, the result will be a Python list containing `Node` objects. XPath expressions that produce string, numeric, or Boolean results will return Python `unicode`, `float`, or `boolean` values respectively.

The *nsMap* argument is optional. If provided, it contains a Python dictionary whose keys are namespace prefixes, and each corresponding value is the namespace URI for that prefix. Namespace prefixes in the XPath expression will be translated to namespace URIs using this dictionary.

Here's an example. Suppose `roster` is an `Element` node. This expression:

```
dl = roster.xpath("player[@status='injured']")
```

¹⁴ <http://www.nmt.edu/tcc/help/pubs/xslt/>

¹⁵ <http://www.w3.org/TR/xpath>

would set `dl` to a list containing all of `roster`'s `player` child elements that have a `status` attribute with the value `'injured'`.

6. The Document class

The node representing the entire document has these attributes:

.baseURI

The document's base URI, or `None` if unknown.

.doctype

At present, 4Suite does not support the attachment of document type information from a `<!DOCTYPE . . .>` declaration. In the future, this attribute may contain the document type as a `DocumentType` node. At present, it will be `None`.

.documentElement

Contains the root element of the document as an `Element` node. For example, if the document is XHTML, this attribute will contain the `html` element.

.publicId

The document's public identifier. If the document was read from a file with a `DOCTYPE` declaration containing a public identifier, that identifier will be stored here.

You may also set this attribute.

.systemId

The document's system identifier. If the document was read from a file with a `DOCTYPE` declaration containing a system identifier, that identifier will be stored here.

You may also set this attribute.

If a document with a system identifier is serialized (see Section 11, "Printing a document" (p. 17)), a `<!DOCTYPE . . .>` declaration will be generated.

If you are not just reading an XML document, but modifying one or generating one from scratch, see Section 12, "Creating a document from scratch: factory methods" (p. 17).

7. The Element class

Objects in class `Element` represent XML elements. Such objects have this attribute:

.nodeName

The qualified name of the element.

Methods on `Element` objects are shown below.

7.1. `Element.getAttributeNS()`

This method is used to retrieve the value of the attribute of a given name of an element node `E`.

```
E.getAttributeNS(nsURI, localName)
```

If `E` has an attribute whose namespace URI matches `nsURI` and whose local name matches `localName`, return the value of that attribute, otherwise return an empty string.

Note

Where methods take an `nsURI` argument, pass the desired namespace URI as an argument.

For the default namespace, pass `None` to the `nsURI` argument. For example, if `E` is an element, this call:

```
E.getAttributeNS(None, "align")
```

returns the value of the `align` attribute in the default namespace, if there is such an attribute.

7.2. Element.getAttributeNodeNS()

Use this method to retrieve an actual `Attr` node, as opposed to an attribute's value.

```
E.getAttributeNodeNS(nsURI, localName)
```

If `E` has an attribute whose namespace matches `nsURI` and whose local name matches `localName`, return that `Attr` node; otherwise return `None`.

7.3. Element.hasAttributeNS()

This predicate tests whether an element `E` has a given attribute.

```
E.hasAttributeNS(nsURI, name)
```

This predicate returns true if `E` has an attribute whose namespace URI matches `nsURI` and whose name matches `name`.

7.4. Element.removeAttributeNS()

Use this method to remove an attribute from an element `E`.

```
E.removeAttributeNS(nsURI, attName)
```

If `E` has an attribute in namespace `nsURI` whose name matches `attName`, and that attribute has a default value, the attribute is replaced by the default value. If there is a matching attribute without a default value, the attribute is removed from `E`.

7.5. Element.setAttributeNS()

This method insures that element `E` has an attribute with a given name and value.

```
E.setAttributeNS(nsURI, qName, value)
```

If `E` has no attribute in namespace `nsURI` and qualified name `qName`, such an attribute is created and set to the given `value`.

If there is an existing attribute by that name, its value is replaced by `value`.

To create an attribute in the default namespace, pass `None` as the first argument, and use an unqualified `qName`.

To create an attribute with a namespace qualifier, pass the namespace URI as the first argument, and use a qualified name of the form "*prefix:localName*". You are responsible for being consistent about the association between a given namespace prefixes and its namespace URI.

7.6. `Element.setAttributeNodeNS()`

Use this to attach an existing `Attr` node *attr* to an element *E*.

```
E.setAttributeNodeNS(attr)
```

8. The `Attr` class

A node of this class represents one attribute of an XML element. Its attributes (in the Python sense) are:

.name

The attribute name.

.value

The attribute's value as a string.

.ownerElement

The `Element` node to which this attribute is attached. If the attribute is an unattached `Attr` node, the `.ownerElement` will be `None`.

9. The `CharacterData` classes: `Text`, `CDATA`, and `Comment`

This class is a base class for nodes that hold strings of text. All strings are represented as 16-bit Unicode characters; use `str()` if you need to coerce them to 8-bit character strings.

There are three classes derived from `CharacterData`: `Text` for text children of elements, `CDATA` for `CDATA` sections, and `Comment` for XML comments.

Its attributes include:

.data

The content of the node as a string.

.length

The length of the content in characters.

Methods defined on `CharacterData` objects are enumerated below.

9.1. `CharacterData.appendData()`

Use this method to append text from a string *s* to a `CharacterData` object *C*.

```
C.appendData(s)
```

9.2. `CharacterData.deleteData()`

Use this method to delete text from a `CharacterData` object *C*.

```
C.deleteData(offset, count)
```

The method removes *count* characters of *C*'s content starting at index *offset*, counting from zero.

The *offset* must be less than the length of the data, or this method will raise an `xml.dom.IndexSizeErr` exception. However, if the *count* extends past the end of the string, all characters through the end of the string will be deleted.

9.3. CharacterData.insertData()

This method inserts additional text from string *s* into the content of a `CharacterData` object *C*.

```
C.insertData(offset, s)
```

String *s* is inserted in *C*'s content starting before the character that was at index *offset*.

The *offset* must be strictly less than the length of the string, or the method will raise an `xml.dom.IndexSizeErr` exception.

9.4. CharacterData.replaceData()

Use this method to replace part of the content of a `CharacterData` object *C*.

```
C.replaceData(offset, count, s)
```

This method replaces *C*'s content starting at index *offset*, for *count* characters, with string *s*.

The *offset* must be less than the length of the existing data, or the method will raise an `xml.dom.IndexSizeErr` exception. The *count* must be at least one or no text will be replaced. If you specify a *count* that extends past the size of the existing data, all characters through the end will be replaced.

9.5. CharacterData.substringData()

To extract part of the textual data of a `CharacterData` object *C*, use this method:

```
C.substringData(offset, count)
```

The method returns the data from *C*'s content starting at index *offset* and continuing for *count* characters.

The *offset* must be less than the length of the data, or the method will raise `xml.dom.IndexSizeErr`. If the *count* extends past the end of the data, the method will return the content of the data up to the end.

10. The DOMImplementation object

In module `Ft.Xml.Domlette`, there is a variable named `implementation` that contains a `DOMImplementation` object. This object contains two methods used to create an entirely new document as a DOM tree.

To get this object, use this form of import:

```
import Ft.Xml.Domlette as domlette
```

Then you can get the `DOMImplementation` object from `"domlette.implementation"`.

These methods both return a new `Document` object.

10.1. `DOMImplementation.createDocument()`

Given a `DOMImplementation` object *I*, this is a convenient method that creates a new `Document` instance and its root `Element` child:

```
I.createDocument(nsURI, qName, docType)
```

where the arguments are:

nsURI

The namespace URI of the root element, if any. See Section 2.6, "Namespace URI" (p. 5). Use `None` if the document is in the default (blank) namespace.

qName

The qualified name of the root element to be created. See Section 2.7, "qName: Qualified name" (p. 6).

This element will be available in the `.documentElement` attribute of the returned `Document` object.

If you are generating a document that uses namespace prefixes, it is your responsibility to be consistent about which namespace prefix corresponds to which namespace URI.

docType

In the official DOM specification, this argument is supposed to be a `DocumentType` object containing the document's public and/or system identifier. However, at present, 4Suite does not support this feature. For now, always pass `None` to this argument.

If you want your document to have a `<!DOCTYPE ...>` declaration, see the `.publicId` and `.systemId` attributes in Section 6, "The `Document` class" (p. 12).

The drawback to this method is that you can't specify the document's base URI (see Section 2.5, "Base URI" (p. 5)). If your document will have relative references in it, see Section 10.2, "`DOMImplementation.createRootNode()`" (p. 16).

10.2. `DOMImplementation.createRootNode()`

Use this method to create a document and specify its base URI (see Section 2.5, "Base URI" (p. 5)). *I* is a `DOMImplementation` object:

```
I.createRootNode(baseURI)
```

where the *baseURI* argument is optional. If provided, this value will be available as the `.baseURI` attribute of the return `Document` node.

This method does not create a root element in the document tree. To create a root element, see Section 12.1, "`Document.createElementNS()`" (p. 18); to link the created element to the document, see Section 5.1, "`Node.appendChild()`" (p. 10).

For more information on creating documents, see Section 12, "Creating a document from scratch: factory methods" (p. 17).

11. Printing a document

To write a DOM Document object to a file in XML form, first import a method like this:

```
import Ft.Xml.Domlette as domlette
```

There are two different serializing (printing) methods.

- Use `.Print()` for correct handling of indenting and line breaks.
- If the output is intended for human consumption, and you don't mind extra line breaks and indentation added for legibility, use `.PrettyPrint()`. Its arguments are the same as the `.Print()` function.

To serialize a document *D*:

Then pass the document object to it using this general form:

```
domlette.Print(D, stream, encoding )
```

where the optional arguments are:

stream

A file object where you want the XML written. Defaults to standard output.

encoding

The character encoding you want to use. The default is 'UTF-8'.

12. Creating a document from scratch: factory methods

Your program can create a complete document as a DOM tree. Here's a general outline for the creation of a document:

1. Start by importing the DOM interface:

```
import Ft.Xml.Domlette as domlette
```

2. Get the DOMImplementation object:

```
dom = domlette.implementation
```

3. Create the Document instance by using the `.createDocument()` method of the `DOMImplementation` object. For the details of this method, see Section 10.1, "DOMImplementation.createDocument()" (p. 16).

For example, suppose you want to create an XHTML page with an `html` root element. This code would do it:

```
doc = dom.createDocument ( None, "html", None )
```

The `Element` object representing that `html` element will now be available in `doc.documentElement`.

4. If you want to attach a document type to the document, assign the system identifier to the Document object's `.systemId` attribute. If there is also a public identifier, store that in the document node's `.publicId`.

For example, suppose you want to declare your document's namespace as XHTML-1.0 Strict. To continue the example code above, where `doc` is a Document object:

```
doc.publicId = "-//W3C//DTD XHTML 1.0 Strict//EN"
doc.systemId = "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
```

If you don't assign a value to `doc.systemId`, no `<!DOCTYPE ...>` declaration will be generated when the document is serialized (that is, converted back to XML).

5. Create the elements, comments, text nodes, and other parts of the document using *factory methods* (see Section 6, “The `Document` class” (p. 12)) on the `Document` object. Connect nodes together by passing each child node to the `.appendChild()` method on the parent node.

To continue the above example of HTML page creation, here is the code to add a `head` element as a child of the `html` element:

```
root = doc.documentElement
head = doc.createElementNS ( None, "head" )
root.appendChild ( head )
```

6. Once you have assembled the tree, print it as described in Section 11, “Printing a document” (p. 17).

12.1. `Document.createElementNS()`

To create a new `Element` node, you must have a `Document` object available.

- If you have a `Node` object available, its `.ownerDocument` attribute will point back to the `Document` object that contains that node.
- To create a new `Document` instance, see Section 10, “The `DOMImplementation` object” (p. 15).

Assuming you have a `Document` instance `D`, use this method to create a new element:

```
D.createElementNS(nsURI, qName)
```

where the arguments have these values:

nsURI

The namespace URI of the element to be created. If the element is in the default namespace, pass the value `None`.

qName

The qualified name of the element. This can be either a local name, or a name of the form `"nsp:localName"` where `nsp` is the namespace prefix corresponding to the `nsURI`.

It is your responsibility to be consistent about the association of namespace prefixes with namespace URIs.

The returned value is a new `Element` node with no parent. To attach a child element `c` to a parent element `p`, use:

```
p.appendChild(c)
```

12.2. `Document.createTextNode()`

To attach text to an element, first build a `Text` node by using the `.createTextNode` constructor on a `Document` instance `D`:

```
D.createTextNode(s)
```

where *S* is the text as a string.

To attach the returned text node *T* to an element node *E*, use:

```
E.appendChild(T)
```

12.3. Document.createProcessingInstruction()

To create a new `ProcessingInstruction` node representing an XML processing instruction with target *t* and data *d*, use this method on a `Document` object *D*:

```
D.createProcessingInstruction ( t, d )
```

The method returns a new `ProcessingInstruction` object with no parent. To attach an unattached node *pi* to a parent node *p*:

```
p.appendChild ( pi )
```

12.4. Document.createComment()

To create a `Comment` node with text content taken from string *s*, call this method on a `Document` instance *D*:

```
D.createComment ( s )
```

The method returns a new `Comment` object with no parent. To attach an unattached node *pi* to a parent node *p*:

```
p.appendChild ( pi )
```

12.5. Document.createDocumentFragment()

Sometimes you want to generate only part of a document. For example, you may want to generate only the `body` element of an XHTML page. If you used a `Document` object, when you printed it you would get an unnecessary `<?xml . . . ?>` declaration at the beginning of the output.

So in this case you should use the `DocumentFragment` constructor on a `Document` object *D*:

```
D.createDocumentFragment()
```

The method returns a `DocumentFragment` object; use this object as you would a `Document` node, as a parent for `Element` nodes. When printed, it will not start with an XML declaration.

12.6. An example of document creation

Here is a Python script that creates a small XHTML file and writes it in XML form to its standard output.

```
#!/usr/local/bin/python
#=====
# creator: Demonstrate building an XML file with a <!DOCTYPE>
#-----
```

```

import Ft.Xml.Domlette as domlette

dom = domlette.implementation

doc = dom.createDocument(None, "html", None)
doc.publicId = "-//W3C//DTD XHTML 1.0 Strict//EN"
doc.systemId = "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"

html=doc.documentElement
doc.appendChild(html)

head=doc.createElementNS(None, "head")
html.appendChild(head)

title=doc.createElementNS(None, "title")
head.appendChild(title)
title.appendChild(doc.createTextNode("Sample title"))

body=doc.createElementNS(None, "body")
html.appendChild(body)

p=doc.createElementNS(None, "p")
body.appendChild(p)
p.appendChild(doc.createTextNode("Sample paragraph text"))
p.setAttributeNS(None, "class", "sample")

domlette.PrettyPrint(doc)

```

Here is the output created by this script:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <title>Sample title</title>
  </head>
  <body>
    <p class="sample">Sample paragraph text</p>
  </body>
</html>

```

13. xml4create.py: A more Pythonic module for XML file creation

The techniques described in the section above, Section 12, “Creating a document from scratch: factory methods” (p. 17), are somewhat clumsy. This clumsiness is due to the DOM’s language-independent charter.

Here, in “literate programming” style, is a small Python module named `xml4create.py` that gives you a more Pythonic tool for XML file creation. For more on literate programming, see *A source extractor for lightweight literate programming*¹⁶.

¹⁶ <http://www.nmt.edu/tcc/help/lang/python/example/litsource/>

For links to the source files developed here, see Section 1.1, “How to get this publication” (p. 3).

Here are the classes exported by the `xml4create.py` module.

13.1. The `DocumentType` class

If your document is to have a `<!DOCTYPE . . .>` declaration attached, you must create a `DocumentType` object with that information before creating your `Document` object, because the document type is an argument to the `Document` constructor.

The constructor has this calling sequence:

```
DocumentType ( rootGI, publicId, systemId )
```

The arguments are:

rootGI

The qualified name of the root element.

publicId

The public identifier, if any, as a string. To get a `<!DOCTYPE rootGI SYSTEM . . .>` document type identifier, pass `None` as this argument.

systemId

The system identifier as a string. Required.

13.2. The `Document` class

To begin creating an XML document, use this constructor to instantiate a `Document` object to hold the content of the document:

```
Document ( rootGI, doctype=None, nsMap=None )
```

Arguments are:

rootGI

The qualified name of the root element.

doctype

A `DocumentType` object specifying the document type. To generate a document with no `<!DOCTYPE . . .>` specifier, pass `None` as this argument.

nsMap

If your document will use namespace prefixes, this argument must contain a Python dictionary that translates namespace prefixes to namespace URIs.

If you are using the default namespace, and you want to include an `xmlns` attribute that links the default namespace to a specific namespace URI, include in your dictionary an entry whose key is `None` and whose value is the namespace URI.

Here is an example of such a dictionary. Namespace prefix `"xsl:"` will refer to the XSLT namespace; prefix `"exsl:"` will refer to the EXSLT (XSLT extensions) namespace; and the default namespace will be XHTML 1.0 Strict.

```
{ "xsl:": "http://www.w3.org/1999/XSL/Transform",  
  "exsl:": "http://exslt.org/common",  
  None: "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" }
```

```
None: "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
}
```

This constructor returns a new `Document` object with one `Element` child whose name is given by `rootGI`.

13.2.1. Document attributes

The `.root` attribute of the return object is the root `Element`.

13.2.2. Document methods

Methods on the `Document` object include:

.doctype

The `doctype` argument that was supplied to the constructor.

.serialize (outFile=None)

Writes the document to a given file. If supplied, `outFile` is a writeable file object. The default output file is `sys.stdout`, the standard output stream.

This method is guaranteed not to add any superfluous whitespace or line breaks to your document. Use it when the document is not intended to be human-readable, because the output may have horrendously long lines.

.write (outFile=None)

This method is similar to `.serialize()`, but it may add line breaks and whitespace for indentation. Use this method to make the output somewhat more human-readable, or when extra whitespace will not change the presentation of the document.

13.3. The Element class

Use this class constructor to add elements to your document:

```
Element ( parent, gi, **attrs )
```

For the root element of your document, pass the `Document` object as the `parent` argument. For all other elements, pass the parent `Element` object as the first argument.

The `gi` argument is the qualified name of the element you are creating (also known as the generic identifier). If it begins with a namespace prefix and colon, that namespace prefix must be a key in the document's `.nsMap`.

You can supply any number of keyword arguments to the constructor, and they will be added as attributes of the new element. For example, if you have an `Element` object named `body`, this code would add a new element as its next or only child:

```
majorHead = Element ( body, "blockquote", align="center" )
```

and the generated XML would look like this:

```
<body>
  <blockquote align="center">
    ...
  </blockquote>
```

```
...
</body>
```

You can add attributes to an existing `Element` object by treating it as a dictionary, storing the new attribute value under a key consisting of the attribute name. For example, another way to do the example above:

```
majorHead = Element ( body, "blockquote" )
majorHead["align"] = "center"
```

Note

You may want to supply attributes whose names are also Python keywords, such as `class=...`. In that case, append an underbar (`_`) to the keyword name, and the `Element` constructor will remove the underbar when building the attribute.

For example, to construct an element that is a child of some element `parent`, that starts with tag `<p class="scream">`, the constructor call would look like this:

```
para = Element ( parent, 'p', class_='scream' )
```

13.3.1. Element methods

`.update (d)`

To add several attributes to an `Element` at once, call this method and pass it a dictionary `d` whose keys are the attribute names, with corresponding attribute values.

13.4. The Text class

To add a text node to an existing parent element `p`:

```
Text ( p, s )
```

The `s` argument is a string containing the text you want to add. For example, suppose you are building an XHTML web page and you have an `Element` object for a paragraph (`p` element) named `helloP`. This code would add an obnoxiously cheerful greeting to the paragraph:

```
Text ( helloP, "Hi! My name is (your name here)!" )
```

13.5. The Comment class

To add a comment as a new child of some element `p`:

```
Comment ( p, s )
```

The second argument is a string containing the text of the comment.

13.6. The DocumentFragment class

If you are building only part of an XML document, use the `DocumentFragment` constructor instead of `Document`:

```
DocumentFragment ( nsMap )
```

where the `nsMap` is a dictionary that maps namespace prefixes to namespace URIs, as in the `Document` constructor.

The constructor returns a new, empty document fragment. It has two methods:

```
.serialize ( outFile=None )  
.write ( outFile=None )
```

Either of these methods will write the content of the fragment to an output file. Its arguments are the same as for the `.serialize()` and `.write()` method of the `Document` class; see Section 13.2.2, “`Document` methods” (p. 22).

Attach content to a `DocumentFragment` the same way you would to a `Document`: by passing it as the first argument to an `Element` or other constructor.

13.7. A small example: XHTML page generation

Here's a small complete script that generates an XHTML web page. The page we want to build looks like this:

```
<?xml version='1.0' encoding='UTF-8'?>  
<!DOCTYPE html PUBLIC "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"  
  "-//W3C//DTD XHTML 1.0 Strict//EN">  
<html>  
  <head>  
    <title>This is the first title.</title>  
  </head>  
  <body>  
    <h1 class='major'>This is the second title.</h1>  
    <p id='a37'>This is the first paragraph.<!--Here's a comment.--></p>  
  </body>  
</html>
```

The script starts with the usual line to make it self-executing, plus an opening comment and the importation of the `xml4create.py` module, which we call `xc` in the script.

```
#!/usr/bin/env python  
#-----  
# xmlcretest: Test driver for xml4create.py  
#-----  
  
import xml4create as xc
```

xmlcretest

Before we can create a `Document`, we need to set up a document type:

```
doctype = xc.DocumentType ( "html", "-//W3C//DTD XHTML 1.0 Strict//EN",  
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" )
```

xmlcretest

Then, we instantiate the `Document` using that document type.

xmlcreatetest

```
doc = xc.Document ( "html", doctype )
```

Attachment of the page's elements proceeds similarly. The first argument to the `Element` constructor is the parent element, and the second argument is the new element's name.

xmlcreatetest

```
head = xc.Element ( doc.root, "head" )
title = xc.Element ( head, "title" )
body = xc.Element ( doc.root, "body" )
```

Adding text content to an element is done by passing it as the first argument to the `Text` constructor:

xmlcreatetest

```
xc.Text ( title, "This is the first title." )
```

Adding attributes to an element can be done by passing keyword arguments to the constructor, like so:

xmlcreatetest

```
h1 = xc.Element ( body, "h1", class_='major' )
xc.Text ( h1, "This is the second title." )
```

You can also add attributes to an element using the syntax for storing into a dictionary:

xmlcreatetest

```
p1 = xc.Element ( body, "p" )
p1['id'] = 'a37'
xc.Text ( p1, "This is the first paragraph." )
```

Adding a comment:

xmlcreatetest

```
xc.Comment ( p1, "Here's a comment." )
```

Finally, write the document to the standard output:

xmlcreatetest

```
doc.write()
```

14. xml4create.py: The source code

Now we move on to the design of the `xml4create.py` module. Design goals include:

- The caller should not have to worry about details of implementation such as which `DOM` variant we're using.
- Regular Python object constructors should be used to create the pieces of an XML document.
- The new module manages access to the objects that contain the factory methods. In order to create a `DOM Element`, you have to have a `DOM Document` element because that's where the `.createElement()` method lives. But to create a `DOM Document` element, you have to have a `DOMImplementation` object, because that's where the `.createDocument()` method lives.

To solve the factory method access problem, the module uses these links between objects so that any node in the tree can get to the factory functions it needs:

- The module's `Document` method contains its `DOM Document` node in its `.node` attribute.
- `Element` objects also have a `.node` attribute that contains the `DOM Element` object.

That DOM Element object has an `.ownerDocument` element that points to its containing DOM Document object.

14.1. Prologue to `xml4create.py`

Now, on to the actual source code of the `xml4create.py` module. First is a prologue that gives the module's documentation string:

```
xml4create.py
"""xml4create.py:  For creating XML files from scratch.

For full documentation, see:
    http://www.nmt.edu/tcc/help/pubs/pyxml4/
"""
```

Next, the imports. We need the `sys` module so we can access the standard output stream, `sys.stdout`. Then we'll need the `domlette` to handle the XML functions.

```
xml4create.py
#=====
# Imports
#-----

import sys
import Ft.Xml.Domlette as domlette
```

14.2. The `DocumentType` class

The purpose of this object is to hold the information necessary to output a `<!DOCTYPE . . . >` declaration.

The 4Suite implementation does not support the DOM's standard `DocumentType` interface. This object is simply a container for the `.publicId` and `.systemId` attributes, so they are available to the `Document` constructor if the user wants to use them. The `rootGI` is also stored away, and if we were picky, we would check it against the `rootGI` argument passed to the `Document` constructor, but that's wasted effort—the latter is always used.

```
xml4create.py
# - - - - - c l a s s   D o c u m e n t T y p e   - - - - -

class DocumentType:
    """Represents an XML document type.

    State/invariants:
        .rootGI:      [ as passed to constructor ]
        .publicId:   [ as passed to constructor ]
        .systemId:   [ as passed to constructor ]
    """
```

The constructor is pro-forma.

```
xml4create.py
# - - -   D o c u m e n t T y p e .   _ _   i n i t   _ _   - - -

def __init__( self, rootGI, publicId=None, systemId=None ):
```

```

        """Constructor for a DocumentType object.

        [ (rootGI is the root element name) and
          (publicId is a public ID as a string or None) and
          (systemId is a system ID as a string) ->
          return a new DocumentType object with those values ]
        """
        self.rootGI = rootGI
        self.publicId = publicId
        self.systemId = systemId

```

14.3. The Document class

Here is the class declaration. Instances contain an attribute named `.node` that will contain the actual DOM Document node.

xml4create.py

```

# - - - - - c l a s s   D o c u m e n t   - - - - -

class Document:
    """Represents an XML document.

    State/invariants:
    .doctype:      [ as passed to constructor, read-only ]
    .nsMap:        [ as passed to constructor, read-only ]
    .dom:          [ the DOMImplementation object ]
    .node:         [ the DOM Document node ]
    """

```

Here's the class constructor:

xml4create.py

```

# - - -   D o c u m e n t .   _ _   i n i t   _ _   - - -

def __init__ ( self, rootGI, doctype=None, nsMap=None ):
    """Constructor for Document.

    [ (domlette is Ft.Xml.Domlette) and
      (rootGI is the root element's qualified name) and
      (doctype is a DocumentType object or None) ->
      return a new Document object with a rootGI Element
      child and document type given by doctype ]
    """

    #-- 1 --
    # [ domlette is Ft.Xml.Domlette ->
    #   self.doctype := doctype
    #   self.nsMap   := nsMap
    #   self.dom     := a DOMImplementation object ]
    self.doctype = doctype
    self.nsMap   = nsMap
    self.dom     = domlette.implementation

```

```

#-- 2 --
# [ if rootGI has a namespace prefix that is not a key in
#   self.nsMap ->
#     raise KeyError
#   else if rootGI has a namespace prefix ->
#     rootNsuri := corresponding namespace URI from
#               self.nsMap
#   else ->
#     rootNsuri := None ]
rootNsuri, rootLocal = self.splitQName ( rootGI )

#-- 3 --
# [ self.dom is a DOMImplementation object ->
#   self.node := a new DOM Document node with default
#               namespace and root element rootGI ]
self.node = self.dom.createDocument ( rootNsuri,
                                     rootGI, None )

```

If the caller gave us a `DocumentType` object, we copy its public and system identifiers over into the DOM Document object.

xml4create.py

```

#-- 4 --
if doctype:
    self.node.publicId = doctype.publicId
    self.node.systemId = doctype.systemId

```

Finally, we create the root node. Here, we take advantage of the polymorphism of the `Element` constructor: it can take a `Document` object as the parent. In that case, instead of creating a new `DOM Document` object, it takes the one found in `self.node.documentElement` and wraps that instead.

xml4create.py

```

#-- 5 --
# [ if rootGI has a namespace prefix that is not a key in
#   nsMap ->
#     raise KeyError
#   else ->
#     self.root := a new Element with qName rootGI ]
self.root = Element ( self, rootGI )

```

14.4. `Document.splitQName()`: Process a qualified name

This utility method is used to convert a qualified name into a namespace URI (if any) and a local name.

xml4create.py

```

# - - -   D o c u m e n t . s p l i t Q N a m e   - - -

def splitQName ( self, qName ):
    """Translate a qualified name into (nsURI, localName)

    [ (self.nsMap is as invariant) and
      (qName is an XML qualified name) ->
      if qName has a namespace prefix ->
        if (self.nsMap is None) or
          (qName's namespace prefix is not a key in nsMap) ->

```

```

        raise KeyError
    else ->
        return (corresponding value from nsMap,
                qName after the namespace prefix)
    else ->
        if None is a key in self.nsMap ->
            return (None, qName)
        else ->
            return (self.nsMap[None], qName) ]
"""

```

First we scan for a colon in the qName. If there isn't one, we're done.

xml4create.py

```

#-- 1 --
# [ if qName contains ":" ->
#     prefix      := qName up to the first ":"
#     localName   := qName after the first ":"
#     else if self.nsMap has a key None ->
#         return (self.nsMap[None], qName)
#     else ->
#         return (None, qName) ]
firstColon = qName.find(":")
if firstColon < 0:
    if self.nsMap:
        try:
            nsuri = self.nsMap[None]
        except KeyError:
            nsuri = None
    else:
        nsuri = None
    return (nsuri, qName)
else:
    prefix      = qName[:firstColon]
    localName   = qName[firstColon+1:]

```

Next we look up the prefix in Document.nsMap. If self.nsMap isn't None, and that prefix is a key in self.nsMap, we return the namespace URI and local name.

xml4create.py

```

#-- 2 --
if self.nsMap is None:
    raise KeyError, ( "Document's root element '%s' had a "
                    "namespace prefix, but no .nsMap was provided." )

#-- 3 --
# [ if prefix is a key in self.nsMap ->
#     nsURI      := the corresponding value
#     else -> raise KeyError ]
nsURI = self.nsMap[prefix]

#-- 3 --
return (nsURI, localName)

```

14.5. Document.serialize()

This method outputs the document tree to the given stream, defaulting to `sys.stdout`.

xml4create.py

```
# - - -   D o c u m e n t . s e r i a l i z e   - - -

def serialize ( self, outFile=None ):
    """Serialize self in XML to outFile.

    [ (domlette is Ft.Xml.domlette) and
      (outFile is a writeable file handle, defaulting to
        sys.stdout) ->
      outFile += a serialized XML representation of self ]
    """

    #-- 1 --
    if outFile is None:
        outFile = sys.stdout

    #-- 2 --
    # [ outFile += an XML rendering of self.node ]
    domlette.Print ( self.node, outFile )
```

14.6. Document.write()

This method outputs the document tree to the given stream, defaulting to `sys.stdout`.

xml4create.py

```
# - - -   D o c u m e n t . w r i t e   - - -

def write ( self, outFile=None ):
    """Prettyprint self in XML to outFile.

    [ (domlette is Ft.Xml.domlette) and
      (outFile is a writeable file handle, defaulting to sys.stdout)
    ->
      outFile += a prettyprinted XML representation of self ]
    """

    #-- 1 --
    if outFile is None:
        outFile = sys.stdout

    #-- 2 --
    # [ outFile += an XML rendering of self.node ]
    domlette.PrettyPrint ( self.node, outFile )
```

14.7. The Element class

As with the `Document` class, instances have a `.node` attribute that points to the contained `DOMElement` node.

```
# - - - - - c l a s s   E l e m e n t   - - - - -

class Element:
    """Represents one XML element.

    State/Invariants:
    .node: [ a DOM Element node containing the actual element ]
    .doc:
        [ the Document element rooting the tree containing
          the parent ]
    """
```

14.8. Element.__init__()

This class wraps a DOM Element object in a more Pythonic class, so that the constructor both creates the new element and links it to its parent.

Normally, this constructor adds a new child element to an existing Element. However, it is also polymorphic: if called with a Document object as its first argument, instead of creating a new DOM Element, it wraps the document's root element.

```
# - - -   E l e m e n t . _ _ i n i t _ _   - - -

def __init__( self, parent, gi, **attrs ):
    """Constructor for Element.

    [ parent is an Element or DocumentFragment object ->
      parent := parent with a new XML element appended
              as its last or only child, having name=gi and
              attributes=attrs
      return that new Element
    parent is a Document object ->
      parent := parent with its .root attribute set to
              a new Element object wrapping
              parent.documentElement, and having
              attributes=attrs ]
    """
```

First we eliminate the special case where parent is a Document.

```
#-- 1 --
# [ if parent is a Document ->
#   self.node := parent.documentElement
#   else if gi is a valid QName in the context of
#   self.doc.nsMap ->
#     parent := parent with a new XML element
#             appended as its last or only child, having
#             name=QName and attributes=attr
#   self.node := a new DOM Element object having
#               the nsURI and localName inferred from
#               self.doc.nsMap
```

```

# else -> raise KeyError ]
if parent.__class__ is Document:
    #-- 1.1 --
    # [ parent is a Document ->
    #     self.node := parent.node.documentElement with
    #         attributes from attrs attached
    #     self.doc := parent ]
    self.__wrapRoot ( parent, **attrs )
else:
    #-- 1.2 --
    # [ if gi is a valid qName in the context of
    #     self.doc.nsMap ->
    #     parent := parent with a new XML element
    #         appended as its last or only child, having
    #         name=gi and attributes from attrs
    #     self.node := a new DOM Element object having
    #         the nsURI and localName inferred from
    #         self.doc.nsMap
    #     self.doc := parent.doc
    #     else -> raise KeyError ]
    self.__newElement ( parent, gi, **attrs )

```

14.9. Element.__wrapRoot(): Wrap the root element

This method handles the special case where the Element() constructor is called to set up the document's root element.

xml4create.py

```

# - - -   E l e m e n t . _ _ w r a p R o o t   - - -

def __wrapRoot ( self, parent, **attrs ):
    """Set up an Element wrapping the document root element.

    [ (parent is a Document) and
      (attrs is a dictionary of attribute names and values) ->
        self.node := parent.node.documentElement with
            attributes from attrs attached
        self.doc := parent ]
    """

```

Since the actual DOM Element has already been created, and lives in the document's .documentElement attribute, all we have to do is set up the class invariants, then copy in any attributes from the attrs argument.

xml4create.py

```

#-- 1 --
self.node = parent.node.documentElement
self.doc = parent

#-- 2 --
# [ if attrs is a dictionary ->
#     self.node := self.node with attribute names
#         (with trailing "_" removed if present) and
#         corresponding values from attrs

```

```

# else -> I ]
if attrs:
    self.update ( attrs )

```

14.10. Element. __setAttr(): Set up one attribute

This method handles storing one attribute name/value pair in the DOM Element object.

xml4create.py

```

# - - -   E l e m e n t .   _ _   s e t   A t t r   - - -

def __setAttr ( self, name, value ):
    """Store one XML attribute in a DOM Element.

    [ name and value are strings ->
      if name has a namespace prefix not defined
      in self.doc.nsMap ->
          raise KeyError
      else if name has a namespace prefix defined
      in self.doc.nsMap ->
          self.node := self.node with a new Attribute
                      added having nsURI=self.doc.nsMap[attrName],
                      localName=(attrName past the first
                      colon, trailing underbar dropped if any),
                      and value=value
      else ->
          self.node := self.node with a new Attribute
                      added having nsURI=None,
                      localName=(attrName, trailing underbar
                      dropped if any), and value=value ]

    """

```

First we have to check to see if the name has a namespace prefix. If so, we translate it to a namespace URI; if there is no prefix, the nsURI will be set to `None`.

xml4create.py

```

#-- 1 --
# [ if name has a namespace prefix ->
#   if that prefix is not a key in self.doc.nsMap ->
#       raise KeyError
#   else ->
#       nsUri      := the corresponding value
#       localName  := name after the colon
#   else ->
#       nsUri      := None
#       localName  := name ]
nsUri, localName = self.doc.splitQName ( name )

```

Next we remove any trailing underbar from name.

xml4create.py

```

#-- 2 --
if name[-1] == '_':
    name = name[:-1]

```

Now we have all we need to create the DOM Element.

xml4create.py

```
#-- 3 --
# [ self.node := self.node with a new attribute added
#     having nsURI=nsUri, name=name, and value=value ]
self.node.setAttributeNS ( nsUri, name, value )
```

14.11. Element.__newElement(): Element child of an element

This method handles the typical case where an element is being added as the child of an existing element.

xml4create.py

```
# - - -   E l e m e n t . _ _ n e w E l e m e n t   - - -

def __newElement ( self, parent, gi, **attrs ):
    """Create an element child of an existing element.

    [ (parent is an Element) and
      (gi is a string) and
      (attrs is a dictionary or None) ->
        if gi is a valid qName in the context of
        self.doc.nsMap ->
            parent      := parent with a new XML element
                          appended as its last or only child, having
                          name=gi and attributes from attrs
            self.node   := a new DOM Element object having
                          the nsURI and localName inferred from
                          self.doc.nsMap
            self.doc     := parent.doc
        else -> raise KeyError ]
    """
```

First, we set up a back-link in `self.doc` so we can jump directly from any `Element` to the containing `Document`.

xml4create.py

```
#-- 1 --
self.doc = parent.doc
```

Next we translate the qualified name into a namespace URI and local name.

xml4create.py

```
#-- 2 --
# [ if gi is a qualified name in the context of
#   self.doc.nsMap ->
#     nsUri      := the namespace URI of gi in that context
#     localName  := the local name of gi
#   else -> raise KeyError ]
nsUri, localName = self.doc.splitQName ( gi )
```

We now have everything we need to create the DOM `Element` node and append it as the next child of the parent.

xml4create.py

```
#-- 3 --
# [ self.node := a DOM Element node with namespace
```

```

#      URI=nsUri and name=name ]
self.node = self.doc.node.createElementNS ( nsUri, gi )

#-- 4 --
# [ parent.node := parent.node with self.node added as
#      its last or only child ]
parent.node.appendChild ( self.node )

```

Finally, we set up any XML attributes if they were supplied.

xml4create.py

```

#-- 5 --
# [ if attrs is None ->
#      I
#      else if attrs contains any namespace prefixes that are
#      not keys in self.doc.nsMap ->
#          raise KeyError
#      else ->
#          self.node := self.node with attribute names
#                      (with trailing "_" removed if present) and
#                      corresponding values from attrs ]
if attrs:
    self.update ( attrs )

```

14.12. Element.__setitem__()

This method allows you to store attributes in an element as if it were a dictionary. For example, if you have an Element object called `e` and you want to an attribute `align='center'`, this would do it:

```
e["align"] = "center"
```

Here's the actual method:

xml4create.py

```

# - - -   E l e m e n t . _ _ s e t i t e m _ _   - - -

def __setitem__ ( self, key, value ):
    """Add an attribute to self.
    [ key is an attribute name as a string ->
      if key has namespace prefix not defined in
      self.doc.nsMap ->
          raise KeyError
      else ->
          self := self with its attribute (key), minus
          any trailing underbar if present, set to (value) ]
    """
    self.__setAttr ( key, value )

```

14.13. Element.update(): Copy XML attributes to the element

The `attrs` argument is a dictionary containing XML attribute names and values. The purpose of this method is to set up those attributes on the DOM Element node.

```
# - - -   E l e m e n t . u p d a t e   - - -

def update ( self, attrs ):
    """Add supplied attributes to the XML Element.

    [ (self.doc is the containing Document) and
      (attrs is a dictionary ->
        if attrs contains any namespace prefixes that are
        not keys in self.doc.nsMap ->
          raise KeyError
        else ->
          self.node := self.node with attribute names
                    (with trailing "_" removed if present) and
                    corresponding values from attrs ]

    """
```

There are two minor complications:

- The attribute name may contain a namespace prefix. We must translate such prefixes into namespace URIs using the `.nsMap` in the containing `Document`. This can fail with a `KeyError` if the attribute's prefix is not in that map.
- If the attribute name ends with an underbar ("`_`"), we remove it. This allows creation of XML attributes whose names are Python reserved words, e.g., `class_` becomes `class`.

All this method has to do is iterate over the elements of the given dictionary, and call `self.__setattr()` for each one.

```
#- - 1 - -
for attrName in attrs:
    #- - 1 body - -
    # [ attrName is a key in attrs ->
    #   if attrName has a namespace prefix not defined
    #   in self.doc.nsMap ->
    #     raise KeyError
    #   else if attrName has a namespace prefix defined
    #   in self.doc.nsMap ->
    #     self.node := self.node with a new Attribute
    #                 added having nsURI=self.doc.nsMap[attrName],
    #                 localName=(attrName past the first
    #                 colon, trailing underbar dropped if any),
    #                 and value=attrs[attrName]
    #   else ->
    #     self.node := self.node with a new Attribute
    #                 added having nsURI=None and
    #                 localName=(attrName, trailing underbar
    #                 dropped if any), and value=attrs[attrName] ]
    self.__setattr ( attrName, attrs[attrName] )
```

14.14. The Text class

These objects hold text nodes.

```
# - - - - - c l a s s   T e x t   - - - - -

class Text:
    """Represents a text node.

    State/Invariants:
        .node: [ a DOM Text node holding the text content ]
    """
```

14.15. Text.__init__()

As with the `Element` constructor, the first argument to this constructor is the `parent` node, whose `.doc` attribute is a `DOM Document` node that carries the `.createTextNode()` factory method we need to create a `DOM Text` node.

```
# - - -   T e x t .   _ _   i n i t   _ _   - - -

def __init__( self, parent, content ):
    """Constructor for Text

    [ parent is an Element object ->
      parent := parent with a new text node added with
              text=content
      return a new Text object representing that text ]
    """

    #-- 1 --
    # [ parent is an Element ->
    #   self.node := a new DOM Text node with content=content ]
    self.node = parent.doc.node.createTextNode ( content )
```

Next, we attach the newly constructed `DOM Text` node to its parent `DOM` node.

```
#-- 2 --
# [ parent := parent with self.node added as its next or
#   only child ]
parent.node.appendChild ( self.node )
```

14.16. The Comment class

There's not much to this class: it stores the content in a `DOM Comment` node.

```
# - - - - - c l a s s   C o m m e n t   - - - - -

class Comment:
    """Represents an XML comment."""
```

```

# - - -   C o m m e n t . _ _ i n i t _ _   - - -

def __init__ ( self, parent, content ):
    """Constructor for a Comment object.
    [ parent is an Element object ->
      parent := parent with a new comment node added with
              text=content
      return a new Comment object representing that node ]
    """

    #-- 1 --
    # [ parent is an Element ->
    #   self.node := a new DOM Comment node with content=content ]

    self.node = parent.doc.node.createComment ( content )

    #-- 2 --
    # [ parent := parent with self.node added as its last or
    #   only child ]
    parent.node.appendChild ( self.node )

```

14.17. The DocumentFragment class

A DocumentFragment object is like a free-floating Element: it can have any number of Element or other children.

xml4create.py

```

# - - - - -   c l a s s   D o c u m e n t F r a g m e n t   - - - - -

class DocumentFragment:
    """Represents part of an XML document.

    State/Invariants:
    .doc:   [ a Document object ]
    .node:
    [ a DOM DocumentFragment object whose .ownerDocument
      attribute contains a DOM Document instance ]
    """

```

The .doc attribute is a DOM Document instance we create strictly because we need its factory functions. No actual content will be added to this Document.

xml4create.py

```

# - - -   D o c u m e n t F r a g m e n t . _ _ i n i t _ _   - - -

def __init__ ( self, nsMap=None ):
    """Constructor for DocumentFragment.
    """

    #-- 1 --
    # [ dom := a DOMImplementation object ]
    dom = domlette.implementation

```

```

#-- 2 --
# [ doc      := a new, empty DOM Document object ]
self.doc = Document('dummy', nsMap=nsMap)

#-- 3 --
# [ doc is a DOM Document object ->
#   self.node := a DOM DocumentFragment object ]
self.node = self.doc.node.createDocumentFragment()

```

14.18. DocumentFragment.serialize()

This is like the .serialize() method in our Document class.

xml4create.py

```

# - - -   DocumentFragment.serialize   - - -

def serialize ( self, outFile=None ):
    """Serialize self in XML to outFile."""
    if outFile is None:
        outFile = sys.stdout
    domlette.Print ( self.node, outFile )

```

14.19. DocumentFragment.write()

This is the same as the .write() method in our Document class.

xml4create.py

```

# - - -   DocumentFragment.write   - - -

def write ( self, outFile=None ):
    """Prettyprint self in XML to outFile."""
    if outFile is None:
        outFile = sys.stdout
    domlette.PrettyPrint ( self.node, outFile )

```

15. Test driver for multiple namespaces: crens

This script, crens, tests the xml4create.py module's facilities for creating elements and attributes in multiple namespaces.

crens

```

#!/usr/bin/env python
#=====
# crens: Namespace output testing for pyxml4.py.
# For documentation, see:
#   http://www.nmt.edu/tcc/help/pubs/pyxml4/
#-----

import xml4create as xc

nsMap = { None: "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd",
          "xsl": "http://www.w3.org/1999/XSL/Transform",

```

```

        "exsl": "http://exslt.org/common" }

doc = xc.Document ( "xsl:stylesheet", nsMap=nsMap )
doc.root['version'] = '1.0'

template = xc.Element ( doc.root, 'xsl:template', match='foo' )

subdoc = xc.Element ( template, 'exsl:document', href='outer.html' )

hr = xc.Element ( subdoc, 'hr' )
hr['exsl:class'] = 'silly'

doc.write()

```

Here's the output (slightly reformatted to fold some long lines):

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
  xmlns:exsl="http://exslt.org/common"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="foo">
    <exsl:document href="outer.html">
      <hr exsl:class="silly"/>
    </exsl:document>
  </xsl:template>
</xsl:stylesheet>

```

16. Test driver for a document fragment: crefrag

This script generates a small Web page without the enclosing `html` element. It demonstrates use of the `DocumentFragment` object, in particular the ability of such an object to have multiple children (unlike a `Document`), and to produce a file with no initial `<?xml . . .>` processing instruction.

crefrag

```

#!/usr/bin/env python
#=====
# crefrag: Fragment output testing for pyxml4.py.
# For documentation, see:
#   http://www.nmt.edu/tcc/help/pubs/pyxml4/
#-----

import xml4create as xc

frag = xc.DocumentFragment()

head = xc.Element ( frag, 'head' )
title = xc.Element ( head, 'title' )
xc.Text ( title, 'No title' )

body = xc.Element ( frag, 'body' )
h1 = xc.Element ( body, 'h1' )

```

```
xc.Text ( h1, 'No body either' )  
frag.write()
```

Here's the output of this script:

```
<head>  
  <title>No title</title>  
</head>  
<body>  
  <h1>No body either</h1>  
</body>
```

